1. **Abstract Data Types**
   1. Define, with appropriate illustrations the following linked data types:
   2. **Ordered (sorted) Linked List** A list in which a key’s i+1 is greater than the key i. It’s ordered.**,**
   3. **Unordered (unsorted) Linked List** A linked list with no particular order to the keys in the sequence.**,**
   4. **Stack** a list in which the first element to be put in is the first element to come out.
   5. **Queue** A linked list in which each item lines up in a queue and becomes the last to be removed. A line basicallyLIFO
   6. **Double Linked List** A list with each node containing two pointers: next and prev, which point to the next and previous node in the list respectively**,**
   7. **Circular Linked List** A linked list in which the next pointer in the tail as a list contains a pointer to the head.
   8. **Circular Double Linked List** A double linked list in which the tail and head of the list are linked by their respective pointers. **.**
2. **List Templates**
   1. Show, using real code, how you would create a Standard Template Library **list** named **m\_InvList** to contain instances of class **InventoryItem**.

list<InventoryItem> m\_InvList;

* 1. Given the instances of class InventoryItem, Itm1, Itm2, and Itm3 show how you would add these to the list.

m\_InvList.push\_back(itm1);

m\_InvList.push\_back(itm2);

m\_InvList.push\_back(itm3);

* 1. Assuming the less than operator (<) has been overloaded in the class show how you would sort the items in the list after all items have been inserted.

m\_InvList.sort();

---------------------------------------------------------

1. **Linear Abstract Data Types**
   1. Linked Lists:
      1. Given the following algorithm for searching a linked list (or a portion of the algorithm) in real code, be able to fill in the blanks.
      2. ListNode \*MyLinkedList::Search(int key)  
         {  
          // Assume ListNode is a structure and contains the variable **int key;**  
          ListNode \*temp = head; **// Search for the key** while((temp!=NULL) && (temp->key!=key))  
          {  
          temp=temp->next; **// Advance to next node** }  
          if(temp !=NULL)  **// Make sure we found the node**  
          { return temp; // **If found return appropriate value**  
          }  
          else   
          { return NULL; **// return NULL when not found**  
          }  
         }
      3. Given the algorithm for inserting new nodes into an ordered linked list (or a portion of the algorithm) in real code, be able to fill in the blanks.
      4. bool MyLinkedList::Insert(ListNode \*newNode)  
         {  
          // Assume ListNode is a structure and contains the variable **int key;**  
          // Assume the function returns true if it successfully inserts the node  
          ListNode \*back = NULL, \*temp = head;  
          if(head == NULL) // Check for inserting first node into an empty list  
          {  
          head=newNode;  
          return true;  
          } else  
          { // Search for insert location  
          while((temp!=NULL) && (temp->key < newNode->key))  
          {  
          back=temp; // Advance to next node  
          temp=temp->next;   
          }  
           
          // Check for inserting at head of the list  
          if(back==NULL)   
          {  
          newNode->next = head; // Insert at head of list  
          head = newNode;  
          return true;  
          }  
          else // Insert elsewhere in the list  
          {  
          newNode->next=temp;  
          back->next=newNode;  
          return true;  
          }  
          }  
          return false; // Should never get here  
         }
      5. Given the algorithm for deleting nodes from an ordered linked list (or a portion of the algorithm) in real code, be able to fill in the blanks.
      6. bool MyLinkedList::Delete(int Key)  
         {  
          ListNode \*back = NULL, \*temp = head;  
          // Search for node to delete  
          while((temp!=NULL) && (temp->key!=key))  
          {  
          back=temp; **// Advance to next node** temp=temp->next;   
          }  
           
          if(temp==NULL) **// Check for node to delete not found** return false;  
           
          else if(back==NULL) **// Check for deleting head of the list** {  
          head=head->next; **// Remove the head of the list** }  
          else   
          {  
          back->next=temp->next; **// Remove node other than at** head  
          }  
          **// Now deallocate the memory used by the removed node** delete(temp);  
          return true;   
         }
      7. Given the algorithm for inserting new nodes into an ordered double linked list (or a portion of the algorithm) in real code, be able to fill in the blanks. Note: There is no back pointer used in this function.
      8. bool MyDoubleLinkedList::Insert(ListNode \*newNode)  
         {  
          **// Assume ListNode is a structure and contains the variable int key;** // and **ListNode \*prev, \*next;**  
          // Assume the function returns true if it successfully inserts the node  
          ListNode \*temp = head;  
          if(head == NULL) // Check for inserting first node into an empty list  
          {  
          head=newNode;  
          return true;  
          } else  
          { // Search for insert location  
          while((temp!=NULL) && (temp->key <newNode->key))  
          {  
          temp=temp->next;   
          {  
           
          // Check for inserting at head of the list  
          if(temp->prev==NULL)   
          {  
          head->prev=newNode;   
          newNode->next=head;  
          head=newNode;  
          return true;  
          }  
          else if((temp->next == NULL) && (newNode->key > temp->key))  
          {  
          // Inserting at the tail  
          temp->next=newNode;  
          newNode->prev=temp;  
          else // Insert elsewhere in the list  
          {  
          // Set newNode's pointers  
          NewNode->prev=temp->prev;  
          newNode->next=temp;  
          // Set next pointer for node before newNode  
          temp->prev->next=newNode;  
          // Set prev pointer for node after newNode  
          temp->prev=newNode;  
          }  
          return true;  
          }  
          return false;   
         }
      9. Given the algorithm for deleting nodes from an ordered double linked list (or a portion of the algorithm) in real code, be able to fill in the blanks.
      10. ListNode \*MyDoubleLinkedList::Delete(int Key)  
          {  
           // Assume ListNode is a structure and contains the variable **int key;**  
           // and **ListNode \*prev, \*next;**  
           // Assume the function returns a pointer to the node if it successfully removes the node  
           // or NULL if the node was not found  
           ListNode \*temp = head;  
           // Search for node to delete  
           while((temp!=NULL) && (temp->key!=key))  
           {  
           temp=temp->next; // Advance to next node  
           }  
            
           if(temp==NULL) // Check for node to delete not found  
           return NULL;  
            
           else if(temp->prev==NULL) // Check for deleting head of the list  
           {  
           head=head->next; // Remove the head of the list  
           if(head != NULL) // Make sure head is not now NULL  
           head->prev=NULL; //Set head's prev pointer  
           return temp; // Return the node removed  
           }  
           else // Delete node elsewhere in the list  
           {  
           temp->prev->next=temp->next; // Move pointer of node before temp  
           if(temp->next != NULL)  
           Temp -> next -> prev = temp -> prev;  
           return temp;  
           }  
           return NULL; // This line will never be reached but it keeps the compiler from complaining   
          }
   2. Stacks:
      1. Define and explain how a Stack works, i.e. describe the abstract data type of a Stack.

A linked list following FIFO. Instead of a head we have a top, we make all new entries the new head, pushing the rest of the information down the line. Thus push and pop.

* + 1. Given the algorithm for the functions of Push() and Pop() in a Stack class in real code, be able to fill in the blanks.
    2. // Define a structure to be used as the stack item  
       struct StackItem  
       {  
        char ch;  
        StackItem \*next;  
       };  
         
       // this stack pushes and pops characters  
       bool MyStack::Push(char ch)  
       {  
        // Create a new node and insert the data  
        StackItem \*newNode = new StackItem();  
        newNode->ch = ch;  
        newNode->next = NULL;  
         
        // Check to see if the stack is empty  
        if(\_\_\_\_\_StackTop==NULL\_\_\_\_\_)  
        {  
        // Push newNode as first in the stack  
        \_\_\_\_\_\_\_StackTop=newNode\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
        }  
        else  
        {  
        // Push newNode on top of the stack  
        \_\_\_\_\_\_\_\_\_\_\_newNode->next=StackTop\_\_\_\_\_\_\_\_\_\_;  
        \_\_\_\_\_\_\_\_\_\_\_StackTop = newNode\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
        }  
        return true; // Signal successful push  
       }  
         
       char MyStack::Pop()  
       {  
        char ch;  
        struct StackItem \*temp;  
         
        // Check for empty stack  
        if(isEmpty()) return '\0'; // Return null character if empty  
         
        // Remove the top item from the stack  
        temp = top;  
        top = \_\_\_\_top->next\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
        // Copy the data from the top item for return  
        ch = temp->ch;  
        // Deallocate the memory used by the removed node  
        \_\_\_\_\_delete(temp)\_\_\_\_\_\_\_\_\_\_;  
         
        // Return the popped character  
        \_\_\_\_return(ch)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
       }
  1. Queues:
     1. Define and explain how a Queue works, i.e. describe the abstract data type of a Queue.

A queue is a linked list in which every item is added to the tail using enqueue and removed from the head using dequeue. It follows LIFO, where each item has to come out in the same order it was put it.

* + 1. Given the algorithm for the functions of Enqueue() and Dequeue() in a Queue class in real code, be able to fill in the blanks.
    2. // Define a structure to be used as the queue item  
       struct QNode  
       {  
        char ch;  
        QNode \*next;  
       };  
       // this queue enqueues and dequeues characters  
       // Assume there are pointers called **head** and  
       // **tail** that point to the first and last  
       // nodes in the queue respectively.  
       bool MyQueue::Enqueue(char ch)  
       {  
        QNode \*newNode;  
         
        // Create new node for the queue  
        newNode= new QNode();  
        newNode->ch = ch;  
        newNode->next = NULL;  
         
        // Check for inserting first node in the queue  
        if(\_\_\_\_head==NULL\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)  
        {  
        head = \_\_newNode\_\_\_\_\_\_\_\_\_\_\_\_\_;  
        tail = \_\_\_newNode\_\_\_\_\_\_\_\_\_\_;  
        }  
        else  
        {  
        \_\_\_\_\_\_tail->next=newNode\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; // Insert into the queue  
        \_\_\_\_tail=newNode\_\_\_\_\_\_\_; // Set tail to new last node  
        }  
         
        return true;  
       }  
         
       char MyQueue::Dequeue()  
       {  
        char ch;  
        QNode \*temp;  
         
        // Check for empty Queue  
        if(\_\_\_\_\_\_head==NULL\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)   
        return '\0'; // Return null character if queue is empty  
        else  
        {  
        ch = \_\_\_\_head->ch\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; // Get character to return  
        // Advance head pointer  
        temp = head;  
        head = \_\_\_\_\_head->next\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
        // Deallocate memory used by the old head  
        delete \_\_\_\_\_\_\_\_temp\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;   
        // Check to see if queue is now empty  
        if(\_\_\_\_\_\_\_\_\_\_\_\_head=NULL\_\_\_\_\_\_\_\_\_\_\_\_\_\_)  
        {  
        head = \_\_\_\_\_NULL\_\_\_\_\_\_\_\_\_\_\_\_; // Reset everything to empty queue  
        tail = \_\_\_\_NULL\_\_\_\_\_\_\_\_;   
        }  
        }  
        return ch;
    3. // Return popped character  
       }

1. **Trees**
   1. Explain, with appropriate diagrams, the structure of the following: (1) Binary Tree, (2) Full Binary Tree, (3) Complete Binary Tree, (3) Heap, (4) AVL tree, (5) 2-3 tree, (6) b-Tree, (7) Trie. Be able to recognize diagrams of each type.
   2. Define the terms related to trees: **root, branches, sub-tree, node, parent, child, descendent, leaf, internal node, tree levels, empty tree**.
   3. Given the following algorithm for searching a binary tree (or a portion of the algorithm) in real code , be able to fill in the blanks.
   4. // Define a structure to be used as the tree node  
      struct TreeNode  
      {  
       int Key;  
       // Other fields defined here  
       TreeNode \*left;  
       TreeNode \*right;  
      };  
      TreeNode \*MyTree::SearchTree(int Key)  
      {  
       TreeNode \*temp = root;  
       // Search for the correct TreeNode  
       while(\_\_\_\_\_temp!=NULL\_\_\_\_\_\_\_\_ && \_\_\_\_\_temp->key!=key\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)  
       {  
       if(Key < temp->Key)   
       temp = \_\_\_\_\_\_temp->left\_\_\_\_\_\_\_\_\_; // Search key comes before this node.  
       else  
       temp = \_\_\_\_temp->right\_\_\_\_\_\_\_\_\_\_\_; // Search key comes after this node   
       }  
       if(temp == NULL) // Check for search key not found  
       return \_\_\_\_\_\_\_temp\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; // Return appropriate value if not found   
       else  
       return\_\_\_\_\_\_\_temp\_\_\_\_\_\_\_\_\_\_\_; // Return apprlpriate value if found  
      }
   5. Given the following algorithm for inserting a node into a binary tree (or a portion of the algorithm) in real code, be able to fill in the blanks.
   6. // Define a structure to be used as the tree node  
      struct TreeNode  
      {  
       int Key;  
       // Other fields defined here  
       TreeNode \*left;  
       TreeNode \*right;  
      };  
      // This function returns true if the node was successfully inserted  
      bool MyTree::Insert(TreeNode \*newNode)  
      {  
       // Create and initialize search pointers  
       TreeNode \*temp = root;  
       TreeNode \*back = NULL;  
       // Search for the location to insert newNode  
       while(\_\_\_temp!=NULL\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)   
       {  
       back = \_\_\_temp\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; // Advance back pointer  
       // Advance temp pointer  
       if(newNode->Key < temp->Key)  
       temp = \_\_\_temp->left\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; else  
       temp = \_\_\_temp->right\_\_\_\_\_\_\_\_\_\_\_;  
       }  
       // Now attach the new node   
       if(\_\_\_back==NULL\_\_\_\_\_\_\_\_\_\_\_\_) // See if inserting this node as the root in a new tree   
       \_\_\_root\_\_\_\_\_\_\_\_ = newNode;  
       else  
       {  
       // Attach as child of existing node  
       if(newNode->Key < back->Key)  
       \_\_\_\_\_\_\_\_back->left\_\_\_\_\_\_\_\_\_\_\_ = newNode;  
       else  
       \_\_\_\_back->right\_\_\_\_\_ = newNode;  
       }  
       return true;  
      }
   7. Given the following algorithm for deleting a a node from a binary tree (or a portion of the algorithm) in real code, be able to fill in the blanks.
   8. // Define a structure to be used as the tree node  
      struct TreeNode  
      {  
       int Key;  
       // Other fields defined here  
       TreeNode \*left;  
       TreeNode \*right;  
      };  
      // This function returns true if the node was successfully deleted  
      bool MyTree::Delete(int Key)  
      {  
       // Create pointers for the search and delete  
       TreeNode \*back = NULL;  
       TreeNode \*temp = root;  
       TreeNode \*delParent; // Parent of node to delete  
      The TreeNode \*delNode; // Node to delete  
        
       // Search for the node to delete   
       while((\_\_\_\_\_temp!=NULL\_\_\_\_\_\_\_\_\_\_\_\_\_) && (\_\_\_\_\_\_temp->key!=key\_\_\_\_\_\_\_))  
       {  
       back = \_\_\_temp\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; // Advance back pointer  
       // Advance temp pointer  
       if(Key < temp->Key)  
       temp = \_\_\_temp=temp->left\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
       else  
       temp = \_\_\_temp=temp->right\_\_\_\_\_\_\_\_\_\_\_\_\_;  
       }  
        
       if(\_\_\_\_\_\_temp==NULL\_\_\_\_\_\_\_\_\_\_\_) // Check to see if the one to delete was NOT found  
       {  
       return false;  
       }  
       else  
       {  
       // Set pointers for next phase of the algorithm  
       delNode = temp; // delNode now points to TreeNode to delete  
       delParent = back; // delParent now points to its' parent which may be NULL  
       }  
        
       // Case 1: Deleting node with no children or possibly one child on left  
       if(delNode->\_\_right\_\_\_\_\_\_\_ == NULL)  
       {  
       if(delParent == \_\_\_NULL\_\_\_\_ ) // Check to see if deleting the root   
       {  
       root = \_\_\_\_\_delNode->left\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
       delete \_\_\_\_\_\_delNode\_\_\_\_\_\_\_\_;  
       return true;  
       }  
       else  
       {  
       // Deleting node other than root  
       if(delParent->left == delNode)  
       delParent->\_\_\_\_left\_\_\_ = delNode->\_\_\_left\_\_\_\_;  
       else  
       delParent->\_\_right\_\_\_\_\_= delNode->\_\_\_left\_\_\_\_;  
       delete delNode;  
       return true;
   9. //Theres one left kid. If the node is the parents lefts child, we set the left parent node to delnodes child. If it’s the rigjht, we set the rightr one to del nodes child.  
       }  
       }  
       else //There is a right child on delNode   
       {  
       if(\_\_\_\_delNode->left==NULL\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) // Check to see if there is a left child  
       {  
       // Case 2: No left child so deleting node with one child on the right  
       if(delParent == \_\_\_\_\_NULL\_\_\_\_\_\_\_) // Check to see if deleting the root   
       {  
       root = \_\_\_delNode->right\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
       delete \_\_\_delNode\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
       return true;  
       }  
       else  
       {  
       // Deleting node other than root  
       if(delParent->left == delNode)  
       delParent->\_\_left\_\_\_\_\_ = delNode->\_right\_\_\_\_\_\_;  
       else  
       delParent->\_right\_\_\_\_\_= delNode->\_\_right\_\_\_\_;  
       delete delNode;  
       return true;  
       }  
       }  
       else // Case 3: Deleting node with two children   
       {  
       // Set pointers to find the replacement value. Locate the node in delNode's   
       // left sub-tree containing the largest key.   
       temp = \_\_\_delNode->left\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;
   10. //Del node’s left child’s rightmost descendant  
        back = \_\_\_\_delNode\_\_\_//the prev step\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
        // Search for the replacement node  
        while(\_\_\_\_\_\_\_\_temp->right\_\_\_\_\_\_\_\_\_\_\_ != NULL)  
        {  
        back = \_\_temp\_\_\_\_\_\_\_\_\_\_\_\_\_;  
        temp = \_\_temp->right\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  
        }this cycles through to the rightmost desc  
         
        // Assume code to copy the replacement values into delNode is here.  
         
        // Remove the replacement node from the tree   
        if(back == \_\_\_\_\_delNode\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) // Is the replacement node delNode's left child?  
        back->\_\_\_left\_\_\_\_\_\_\_\_\_ = temp->\_\_\_left\_\_\_\_\_\_\_\_\_\_\_; // Yes it is  
        else  
        back->\_\_\_\_right\_\_\_\_\_ = temp->\_\_\_\_left\_\_\_\_\_\_; // No, it's a descendent of left child //It’s either delNodes left child or a rightmost descendant of delNodes left child that you set this new copy nodes pointer to.  
        delete temp;  
        return true;  
        }  
        }  
       }
   11. List the three different node arrangements (cases) and the two variations of each that are possible when deleting a node from a binary tree. Draw a diagram illustrating each case.

First case: The node has one left child or no children. (if delNode==right = NULL;)

Variation 1: Back==NULL thus it is the root we’re deleting

root=delNode->left;

Varation 2: It’s a descendant node we’re deleting.

We then check if delParent’s left=del node in which case we set delParent’s left to delNode’s left. Else, we set del parents right to del nodes left. Makes sense.

Second Case: delNode has one right child

Variation 1: The node is the root, in which case root=delNode->right;

Variation 2: It’s not the root. We see if delParent->left=delNode, if it is, we set delParent’s left to delNodes right. Else we set delParent’s right to del node’s right.

* 1. Show how you would perform a recursive traversal of a binary tree.

searchTree(\*root)

{

while(root!=NULL)

{

searchTree(root->right)

searchTree(root->left)

}

}

* 1. Explain how a heap can be used as a priority queue.

NO idea. I guess because no child can have a key above it’s parent, it means that the heap is organized perfectly according to key and it can control what nodes are more or less important.

* 1. Explain how AVL trees are kept in near perfect balance, i.e. the 4 types of rotations.

IF the tree is out of balance to the right by two levels you perform a single left rotaton by taking the ancestor out of balance and setting it’s right child’s left child equal to it’s right child. We the make The parent node the left child of it’s right child in order to make it balanced.

FOr a single right rotation, the ancestor is out of balance to the left. We would make it’s left child’s right child the right child of the parent, we then make the right child the new parent and cause the old parent to become the new right child of B. it’s called that because we rotate every thing to the right and the relatrionships are kept the same.

We have three pointers in the case of the first two rotations. A left pointer, a right pointer, and a parent pointer.

You rotate by making the right childe left pointer the new parent left, then you make the parent the left child of it’s previous right child.

Right rotation you make the left childs right pointer equat to the left child of the parent, then makeing the parent the right child of it’s left.